Apple Assembly Line

$1.80

Volume 7 -- Issue 6 March, 1987

In This Issue...

Sorry we haven't written sooner...

This really is the March issue you're receiving sometime in the middle of April, and there's no blaming the Post Office. We've had a surplus of other work lately and a shortage of AAL inspiration. (That's a hint: you can send in articles and programs and help us and all AAL readers. We are especially interested in good short items about exploring the IIgs.) We were almost back on track by last December, but slipped again. We expect to catch up across the next several months....

About those Disk Subscriptions

When you subscribe to the Apple Assembly Line Monthly Disks you not only receive all the source code from an issue, you now also get the text of all that month's articles (in DOS 3.3 Text files.) We know of six blind subscribers who really appreciate this format, and there may be more of you who are interested. Our thanks to Larry Skutchan, of the American Printing House for the Blind, for prompting us to add this service.

Some Bugs in Apple's ProDOS Version 1.3...Bob Sander-Cederlof

Apple recently released version 1.3 of ProDOS, and sent copies to all licensees. We started sending it out with the ProDOS version of the S-C Macro Assembler. But last week (around March 16th) we received a letter from Apple "recalling" version 1.3. It has two serious problems.

First, there is a "BRA" (BRanch Always) opcode in it. This means version 1.3 will not operate in an older Apple with only a 6502 microprocessor. If you have a 65C02 or 65802 or 65816, no problem here. You are safe in a IIgs, //c, or enhanced //e. You are also safe if you have upgraded the cpu chip yourself in an older machine. That offending instruction could just as well be changed to a BEQ opcode, because that would always branch in this case. With that change the 6502 machines work fine.

Second, when the Apple experts tried to implement the patch developed in Australia and reported first by Tom Weishaar in "Open Apple" to fix an elusive disk-trashing problem, they didn't do it right. This is the same fix I reported in the November AAL, page 13. (Turns out I didn't report it right either, because I overlooked one part of the patch. More on this below.) The way Apple did it causes severe problems with two-drive systems. When you are accessing drive two, version 1.3 keeps switching back to drive 1. If you try to use FILER to copy a volume from drive 1 to a blank disk in drive 2, and if you remember to write-protect the disk in drive 1, FILER will hang up with an I/O ERROR after initializing the drive 2 disk. FILER evidently applies the drive 1 write-protect status to drive 2, and gives up. I don't even want to experiment without having drive 1 write-protected! On the other hand, if you copy from drive 2 to drive 1 it works, but it takes a lot longer than it should to read each segment from the source disk in drive 2.

Both bugs can be fixed by rather simple patches. Boot up PRODOS, into either BAISC.SYSTEM or the S-C Macro Assembler. Then load the PRODOS file, with "BLOAD PRODOS,TSYS,A$2000". Then get into the monitor with "MNTR" from S-C Macro Assembler or "CALL-151" from BASIC.SYSTEM. Type the following patches:

```
        4CCD:F0          (was 80, changing BRA to BEQ)
        5204:BD 8E C0    (was EA EA EA)
        58C3:BD 80 C0 BD 82 C0 BD 84 C0 BD 86 C0
```

Then get back into the system by typing "3D0G", and save the new version with "BSAVE PRODOS,TSYS,A$2000".

The third patching line above replaces Apple's flawed loop which walked on too many soft-switches. Apple's loop does a LDA from C080, C082, C084, and C086; this is correct. It also does it from C088, C08A, C08C, and C08E. This is not correct. It turns off the motor and selects drive 1. The only correct one among this group of four was C08E, intended to be sure the selected drive is in read mode.

S-C Macro Assembler Version 2.0......DOS $100, ProDOS $100, both for $120
Version 2.0 DOS Upgrade Kit for 1.0/1.1/1.2 owners..................$20
ProDOS Upgrade Kit for Version 2.0 DOS owners........................$30
Source Code of S-C Macro 2.0 (DOS or ProDOS)........each, additional $100
S-C Disassembler (ProDOS only)...without source code $30, with source $50
RAK-Ware DISASM (DOS only).......without source code $30, with source $50
ProVIEW (ProDOS only)..$20
Full Screen Editor for S-C Macro (with complete source code)..........$49
S-C Cross Reference Utility......without source code $20, with source $50
S-C Word Processor (with complete source code)........................$50
DP18 and DPFP, including complete source and object code..............$50
ES-CAPE (Extended S-C Applesoft Program Editor)..........
Including Version 2.0 With Source Code..........$50
ES-CAPE Version 2.0 and Source Code Update (for Registered Owners)....$30
Bag of Tricks 2 (Quality Software).........................($49.95) $45 *
MacASM -- Macro Assembler for MacIntosh (Mainstay)..........($150.00) $50 *
S-C Documentor (complete commented source code of Applesoft ROMs).....$50
Cross Assemblers for owners of S-C Macro Assembler.....$32.50 to $50 each
(Available: 6800/1/2, 6301, 6805, 6809, 68000, Z-80, Z-8, 8048,
8051, 8085, 1802/4/5, PDP-11, GI1650/70, others)
Beagle Bros. Applesoft Compiler (ProDOS only)...............($74.95) $65 *

AAL Quarterly Disks.........................each $15, or any four for $45
Each disk contains the source code from three issues of AAL:
Jan-Mar, Apr-Jun, Jul-Sep, and Oct-Dec.
(All source code is formatted for S-C Macro Assembler. Other assemblers
require some effort to convert file type and edit directives.)

Diskettes (with hub rings)........................ package of 25 for $15 *
Vinyl disk pages, 6"x8.5", hold two disks each.................10 for $6 *
Diskette Mailing Protectors (hold 1 or 2 disks).............40 cents each
(Cardboard folders designed to fit 6"X9" Envelopes.)   or $25 per 100 *
Envelopes for Diskette Mailers......................... 6 cents each

65802 Microprocessor (Western Design Center)..................($95) $50 *
quikLoader EPROM System (SCRG)...............................($179) $170 *
PROmGRAMER (SCRG)..($149.50) $140 *
Minuteman 250 Uninterruptible Power Supply...................($359) $320 +

"Apple //gs Technical Reference", Fischer..................($19.95) $19 *
"65816/65802 Assembly Language Programming", Fischer........($21.95) $20 *
"Programming the 65816", Eyes...............................($22.95) $21 *
"Apple //e Reference Manual", Apple Computer................($24.95) $23 *
"Apple //c Reference Manual", Apple Computer................($24.95) $23 *
"ProDOS Technical Reference Manual", Apple Computer.........($29.95) $27 *
"Apple ProDOS: Advanced Features for Programmers", Little..($17.95) $17 *
"ProDOS Inside and Out", Doms & Weishaar...................($16.95) $16 *
"DOStalk Scrapbook", Weishaar & Kersey.....................($14.95) $14 *
"Beneath Apple ProDOS", Worth & Lechner....................($19.95) $18 *
"Beneath Apple DOS", Worth & Lechner.......................($19.95) $18 *
"Inside the Apple //c", Little.............................($19.95) $18 *
"Inside the Apple //e", Little.............................($19.95) $18 *
"Understanding the Apple //e", Sather......................($24.95) $23 *
"Understanding the Apple II", Sather.......................($22.95) $21 *
"Apple II+/IIe Troubleshooting & Repair Guide", Brenner.....($19.95) $18 *
"Apple][Circuit Description", Gayler.....................($22.95) $21 *
"Assem. Language for Applesoft Programmers", Finley & Myers.($18.95) $18 *
"Now That You Know Apple Assembly Language...", Gilder......($19.95) $18 *
"Enhancing Your Apple II, vol. 1", Lancaster...............($15.95) $15 *
"Enhancing Your Apple II, vol. 2", Lancaster...............($17.95) $17 *
"Assembly Cookbook for the Apple II/IIe", Lancaster.........($21.95) $20 *
"Microcomputer Graphics", Myers............................($14.95) $14 *
"Assembly Lines -- the Book", Wagner.......................($19.95) $18 *
"AppleVisions", Bishop & Grossberger.......................($39.95) $36 *

* On these items add $2.00 for the first item and
$.75 for each additional item for US shipping.
+ Inquire for shipping cost.
Foreign customers inquire for postage needed.
Texas residents please add 7.25 % sales tax to all orders.

*** S-C SOFTWARE, P. O. BOX 280300, Dallas, TX 75228 ***
***   (214) 324-2050   ***
*** Master Card, VISA, Discover and American Express ***

```
 Old Code           Apple's Loop         My Patch
-----------       ---------------      -----------
STA $C080,X          LDY #8            LDA $C080,X
STA $C082,X       .1 LDA $C080,X       LDA $C082,X
STA $C084,X          INX               LDA $C084,X
STA $C086,X          INX               LDA $C086,X
                     DEY
                     BNE .1
                     NOP
                     NOP
```

My patch puts the C08E load where Tom's Australian-connection originally put it, over some NOPs which immediately followed the JSR to the code shown above.

Now about my incomplete patch to version 1.1.1 from last November. I omitted the "LDA $C08E,X", which gets patched at location $5004 in this version. I also mis-typed the address for the other patches as going at $56D3 when they actually belong at $56C3. So, in version 1.1.1, following the same load-patch-save sequence above, the patches are:

```
       5004:BD 8E C0    (was EA EA EA)
       56C3:BD 80 C0 BD 82 C0 BD 84 C0 BD
               (changing 9D's to BD's)
```

Version 1.4 is due out soon, and we trust Apple will have it all right this time. Still, I am getting skittish.

Meanwhile, unless you are using a IIgs, you may wish to stick with version 1.1.1. The only real advantage the newer versions have is automatic recognition of the IIgs clock-calendar chip. You don't need this feature if you are not running in a IIgs.

Notes on the IIgs Monitor................Sandy Mossberg

Bob's article "New Features in Apple //gs Monitor" in the January 1987 issue of AAL was interesting and useful, but I have several corrections. I also have two nice examples for the U-command, which allows you to call tools from the monitor.

First, I noticed Bob said that the Change Cursor Command is Control-6 followed by the new cursor character. While that works on his keyboard, the actual character is Control-Shift-6. Apparently Bob's IIgs, which is a //e box and keyboard with a IIgs motherboard, accepts Control-6 and Control-Shift-6 to generate the same code ($9E). In "true" ASCII, this key is really known as Control-Shift-N. If you remember the old Apple II or II+ keyboards, which were modeled after teletypes, the carat character (ASCII $DE) was generated by a Shift-N; likewise, a Control-Shift-N generated the ASCII code $9E. On the //e, //c, and IIgs the carat has been moved to the Shift-6 position. Several other characters were moved at the same time, allowing the all of the alphabet keys could use the Shift Key to control lower- and upper-case input.

Second, Bob apparently misunderstood the meaning of the Flip ASCII capability. When you are entering data into RAM with the "addr:" command, or specifying patterns for the "\pattern\<addr.addrP" search command, you can enter ASCII strings or flip ASCII strings.

Normal strings are entered by typing any number of characters between quotation marks. The status of the ASCII FILTER MASK determines whether the string is low-ASCII or high-ASCII. You control the filter mask using the monitor "=" command: type "7F=F" before entering strings you want in low-ASCII, or "FF=F" before strings in high-ASCII.

Flipped strings can be from one to four characters. The characters are typed between apostrophes, and will be stored in RAM or used as a pattern in REVERSE ORDER. Again, the characters will be in either low- or high-ASCII depending on the state of the filter mask. For example:

```
*FF=F
*300:"ABCDEFGH"
*300.307
00/0300:C1 C2 C3 C4 C5 C6 C7 C8-ABCDEFGH

*7F=F
*300:"ABCDEFGH"
*300.307
00/0300:41 42 43 44 45 46 47 48-ABCDEFGH

*300:'ABCD' 'EFGH'     (note filter still 7F)
*300.307
00/0300:44 43 42 41 48 47 46 45-DCBAHGFE

*FF=F 300:'ABC' 'MNO' 'ST'
*300.307
00/0300:C3 C2 C1 CF CE CD D4 D3:CBAONMTS
```

The monitor "U" command gives you the ability to call tools in the various tool sets. In particulary, I have worked out the method for calling the two time routines in the Miscellaneous Tool Set.

Call ReadTimeHex (tool set $03, function $0D)

```
*\08 08 00 00 00 00 00 00 00 00 0D 03\U

Tool Error-> 0000
06 1F 17 57 1B 02 00 07
```

Interpretation of Input:

```
08 = 8 bytes on stack
08 = 8 bytes off stack
00...00 = 8 zero bytes constitute parms
0D = function number in toolset
03 = toolset number
```

Interpretation of Output:

```
Error 0000 means NO ERROR
06 = 6 seconds
1F = 31 minutes
16 = 22 hours
57 = year 1987
17 = 24th day (0=first day)
02 = 3rd month (0=first month)
00 = null
03 = 3rd day of week (1=Sunday, first day)
```

Call ReadASCIITime (tool set $03, function $0F):

```
*\04 00 00 00 03 00 0F 03\U 300.317

Tool Error-> 0000
00/0300:A0 B4 AF A0 B1 AF B8 B7: 4/ 1/87
00/0308:A0 A0 B9 BA B1 B3 BA B0:  9:13:0
00/0310:B3 A0 C1 CD FF 00 00 FF:3 AM....
```

Interpretation of Input:

```
04 = 4 bytes on stack
00 = 0 bytes off stack
00 00 03 00 = buffer address at 00/0300
0F = function number in toolset
03 = toolset number
```

Interpretation of Output:

```
Error 0000 means NO ERROR
300.313 has string " 4/ 1/87  9:13:03 AM"
```

Beagle Compiler Review............................Bill Morgan

Ever since ProDOS first appeared in 1983 people have been asking us about an Applesoft compiler that would run under that system, but we've had to tell them that there was no such thing. Well the good folks at Beagle Bros have remedied that situation by bringing us The Beagle Compiler, by Alan Bird.

This isn't a true native code compiler, it's more of a very high-speed interpreter. As a result, the compiled program is usually smaller than the original source Applesoft program. On the other hand you lose some memory because of the extra space taken by the compiler's runtime system. The compiler and runtime system together take about 11K, but you can cut that figure in half if you only need the runtime portion. If your program will not fit the first time you try, you can probably take advantage of the compiler's ability to use //e or //c auxiliary memory for storage of arrays and strings.

Beagle claims a 2 to 15 times speedup of a program, depending on what operations are most common. They say that string and variable manipulations show the most improvement, while heavy floating point calculations aren't affected at all. The compiled code automatically uses integer arithmetic whenever possible, thereby avoiding a lot of floating point arithmetic in most programs.

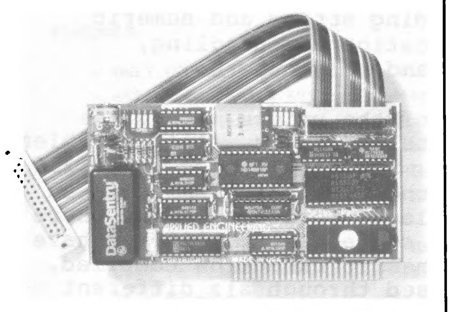
There are several programs in the package: COMPILER.SYSTEM is the normal runtime interpreter, replacing and extending BASIC.SYSTEM. COMPILER processes an Applesoft program file into its high-speed version. AUX.SLOT.SYSTEM & APPLEMEM.SYSTEM (included with version 2.0 and beyond) are versions of COMPILER.SYSTEM that take advantage of memory expansion cards for array and string storage. You have to have one of the SYSTEM files to execute a compiled program, but you can distribute these files with your product "after paying a very reasonable licensing fee."

The Beagle Compiler is made to be very simple to use. After booting into the SYSTEM your STARTUP program can execute COMPILER. Then you just need to type RUN <filename> to load your file, compile it, and start execution. Once you have a finished program you can type COMPILE <source file>,<object file> to produce a disk file of the compiled code. To execute that file you no longer need COMPILER, just one of the SYSTEM files.

As supplied, the compiler uses the standard Applesoft INPUT, but they also include an "INPUT anything" routine that allows commas and semicolons. There is also a FAST.HPLOT routine available that noticeably speeds up HPLOT, but at the cost of an extra 1K of memory for the runtime code. A special SLOW.PDL patch can slow down game paddle reading so you always get the correct reading.

This compiler, unlike other Applesoft compilers we've seen, allows full use of &-routines, including parameter passing. An &-call without parameters will work just fine as it's written;

Serial Pro®
The intelligent multifunction card from Applied Engineering.



Serial Pro is a powerful multifunction card for the Apple IIe, II+, IIGS and compatible computers. Serial Pro combines a powerful serial port for interfacing to a printer, modem or other serial device, with a full function clock/calendar. Serial Pro's serial port is compatible with virtually all letter quality and dot matrix printers, including Apple's Imagewriter and Imagewriter II, and can communicate with the fastest external modems at baud rates from 50 to 19,200.

Set up is easy, because unlike other cards, there are no switches to program serial parameters or disks to use. Serial Pro has a built-in setup screen that allows all parameters to be set. And on-screen help that makes it super-easy. An on-board battery stores your selections for over 20 years, but you can change them at any time. Serial Pro is the perfect upgrade for Super Serial card owners looking for more performance or a built-in clock.

Serial Port Features • Fully Super Serial Card compatible but more versatile and easier to use • Compatible with printers • Compatible with modems • Prints graphics to printer (without software) in normal mode, 90° rotation, double size, side by side, and black/white inversion • Variable word length, number of stop bits and parity selection • Crystal controlled circuits are used to provide ultra accurate baud rates.

Clock Calendar Features • ProDos, DOS and AppleWorks compatible. Displays time and date on the AppleWorks screen • Automatically time and date stamps AppleWorks files • 20 year auto recharging nickel-cadmium battery • Realtime interrupts • Built-in on screen time and date settings (no software needed) • Does not use phantom slots.

Serial Pro is perfect for Apple IIe, II+, IIGS or compatible owners in need of a serial port or clock/calendar or both. But unlike other multifunction cards on the market, Serial Pro does not use phantom slots, so all the slots in your computer are useable.

Serial Pro comes complete with manual, cable and instructions for connecting to all the leading printers and modems.

Serial Pro **$159**

Order Today! See your dealer or call Applied Engineering, 9 a.m. to 11 p.m. 7 days. Or send check or money order to Applied Engineering. MasterCard, VISA and C.O.D. welcome. Texas residents add 5⅛% sales tax. Add $10.00 outside U.S.A.

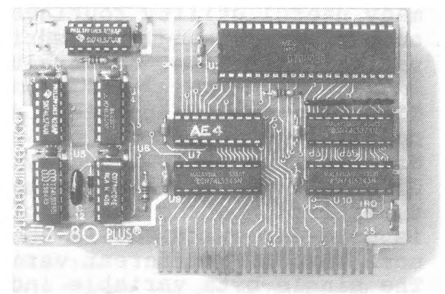
APPLIED ENGINEERING™

The Apple enhancement experts.

(214) 241-6060

P.O. Box 798, Carrollton, TX 75006

With the Z-80 Plus®, run over 5000 new CP/M® programs.



Now, get two computers in one, and all the advantages of both.

Enter the CP/M world with the new Z-80 Plus card from Applied Engineering, and introduce your Apple IIe, IIGS or II+ to thousands of CP/M programs. Only the Z-80 Plus comes standard with the new 5.1 software, the most advanced system ever for running CP/M programs.

The new 5.1 boasts advanced features like built-in disk emulation for popular memory expansion boards, boosting both system speed and storage capacity. And menu-driven utilities that let you get to work faster. The Z-80 Plus also lets you run older CP/M programs—all the way down to Version 1.6 (2.2 is the most popular).

The Z-80 Plus is the only card on the market capable of accessing more than 64K. If you have an extended 80-column card, all 128K is usable. And if you have RamWorks, RamFactor, GSRam, GSRam Plus, or an Apple memory card in your IIe, IIGS or II+, up to 1088K is available.

Each Z-80 Plus comes with our CP/M Ram Drive software, enabling most memory cards to work as a high-speed Ram disk so CP/M software can run up to *twenty times faster.* So packages like WordStar, dBASE II, Turbo Pascal and most others run at blinding speed.

Simply plug the Z-80 Plus into any slot in your Apple. You'll get the benefits of two computers in one—all at an unbelievably low price (only $149!).

• Fully compatible with ALL CP/M software • Fully compatible with 5¼" *and* 3½" disk drives as well as most hard disks including the Sider • Fully compatible with Microsoft disks (no pre-boot required) • Specifically designed for high speed operation • Runs WordStar, dBASE II, Turbo Pascal, and ALL other CP/M software with *no pre-boot* • Semi-custom I.C. and low parts count allows Z-80 Plus to fly through CP/M programs with extremely low power consumption (we use the Z-80B) • Does EVERYTHING other Z-80 boards do, *plus* Z-80 interrupts • Five-year warranty

To get your Z-80 Plus or for more information see your dealer, or call 214-241-6060 9 a.m. to 11 p.m. seven days, or send check or money order to Applied Engineering. MasterCard, VISA and C.O.D. welcome. Texas residents add 5⅛% sales tax. Add $10.00 outside U.S.A.

Applied Engineering™
P.O. Box 798, Carrollton, TX 75006
(214) 241-6060

CPM is a registered trademark of Digital Research

one with parameters will need to be rewritten to match the compiler's syntax and evaluation needs, but the new procedures are easy to use and very well documented. It's not immediately clear how well an "&-interpreter" will work with this approach.

Beagle does an excellent job of documenting how a machine language program can get along with a compiled program. They list all the keyword entry points as well as the entries to many useful utility routines, including string and numeric output, error handling, string allocation and handling, variable assigment and evaluation, and many others.

They also provide complete details of the unique method the compiler uses to store and access its variables. All variables are referred to by a one byte index, so there can be up to 256 different variables or arrays. By the way, this really is an upper limit on the number of variables. For some reason there is no clear statement in the manual saying so, but if you have more than 256 different variable names it cannot be compiled. The single-byte variable index is used through six different pointers to access the values. The six pointers are:

```
     $78  Variable Type (array, string, numeric, FN, integer)

     $7A  Val1 -- array: LSB of header address
                  string: LSB of string address
                  FP: non-zero flags FP, 1st byte of value
                  integer: a zero here flags integer

     $7C  Val2 -- array: MSB of header address
                  string: MSB of string address
                  FP: 2nd byte of value
                  integer: LSB of value

     $7E  Val3 -- array: LSB of array address
                  FP: 3rd byte of value
                  integer: MSB of value

     $80  val4 -- array: MSB of array address
                  FP: 4th byte of value

     $82  Val5 -- array: number of dimensions, 0 if
                  undimensioned
                  FP: 5th byte of value
```

This approach to dealing with variables is one of the main keys to the Beagle Compiler's speed. Another great factor in the speed is that GOTO and GOSUB can point to absolute addresses in RAM, rather than to a line number that must be searched for during execution.

Here's an example of a short Applesoft program along with a hex dump of its compiled result. (When COMPILER.SYSTEM finishes executing a program it trashes its pointers, so that's why I ended my little program with a CALL-151 to get us into the Monitor with everything intact.)

```
10 D$ = CHR$(4):BUFFER = 4096:REM $1000
20 ARGUMENT = 20
30 RESULT = BUFFER / 2 + ARGUMENT * 3.3
40 MYSTRING$ = "AARDVARK"
50 CALL -151

0078-43 08 48 08 4D 08 52 08 57 08 5C 08

0800-00 00 00 11 56 8D 05 40 00 42 00 00 00 00 00 00
0810-00 34 00 B6 94 04 14 01 96 00 10 14 02 94 20 14
0820-03 A8 AE 9C 01 94 02 AC 9C 02 98 82 53 33 33 33
0830-34 04 9A 08 41 41 52 44 56 41 52 4B 6E 96 69 FF
0840-12 FF 12 40 00 00 00 40 FE 00 00 8C 33 65 00 20
0850-06 08 00 10 00 99 00 00 00 00 99 00 00 00 00 9A
0860-00
```

The compiled code for the program apparently begins at $0810, and continues to $0842. In comparison, the original Applesoft program (not counting 35 bytes used to store the variables) ran from $0800 through $0875. There are five variables, so the six tables which begin at $0843 are each five bytes long. Using the table which comes in the Compiler manual I was able to "decompile" the compiled code as follows:

```
00     Initialize and run program

       D$=CHR$(4)
34     Assign value to simple string variable
00     1st Variable (D$)
B6     CHR$
94 04  One-byte integer (04)

       BUFFER=4096
14     Assign value to simple numeric variable
01     2nd Variable (BUFFER)
96 00 10  Two-byte integer ($1000=4096)

       ARGUMENT=32
14     Assign value to simple numeric variable
02     3rd variable (ARGUMENT)
94 20  One-byte integer ($20=32)

       RESULT = BUFFER/2 + ARGUMENT*3.3
14     Assign value to simple numeric variable
03     4th variable (RESULT)
A8     addition    [Notice the Polish form here]
AE        division
9C 01        get value of 2nd variable (BUFFER)
94 02        One-byte integer (2)
AC        Multiplication
9C 02        get value of 3rd variable (ARGUMENT)
98           Five-byte floating point constant follows
82 53 33 33 33   Floating point constant (3.3)
```

```
        MYSTRING$="AARDVARK"
34 04   Assign value to 4th variable (MYSTRING$)
9A      A string constant follows...
08 41 41 52 44 56 41 52 4B   String, 8 bytes, "AARDVARK"

        CALL -151
6E       Call
96 69 FF   Two-byte integer ($FF69 = -151)

12       End of program
```

Notice how all variables are referenced by the one-byte index, and how the constants are compiled in-line in their final form (integer, floating-point, or string). Looking at the six variable slices, and lining them up for easier viewing, I see:

```
00 -- 40 FE 65 00 00 00   D$ = CHR$(4)
01 -- 00 00 00 10 00 00   BUFFER = 4096
02 -- 00 00 20 00 00 00   ARGUMENT = 32
03 -- 00 8C 06 99 99 9A   RESULT = 2153.6
04 -- 40 33 08 00 00 00   MYSTRING$ = "AARDVARK"
```

Maybe all this detail is overkill, but I wanted to show that the Beagle Compiler is real, well-constructed, valuable, and interesting. It is an excellent value at their price of $74.95. If you remember, the various DOS-based compilers cost more and (many would agree) delivered less. We like this one enough to help you get a copy: you can buy them through us for $65.

Commented Listing of ProDOS, $DEF3-DFE4...Bob Sander-Cederlof

As I promised last December, here is another piece of ProDOS. This time I am unveiling the code which processes IRQ interrupts, and the handlers for the related MLI calls. All of the following applies to version 1.1.1 of ProDOS. Later versions may differ in this area.

The two MLI calls related to interrupts are $40 (Allocate Interrupt) and $41 (De-Allocate Interrupt). There is room inside ProDOS for connecting up to four user-coded routines for processing IRQ interrupts. The Allocate Interrupt call stores the address of your routine at the next available entry in the IRQ Path Table. This table exists in the MLI Global Page ($BF00-BFFF), and is shown in lines 1140-1170 in the listing below. When you boot ProDOS these four entries all contain $0000, indicating no interrupts are allocated. An MLI call of the form:

```
       JSR $BF00
       .DA #$40,IRQ.IOB
```

with an IOB like this:

```
       IRQ.IOB .DA #2
       IRQ.NUM .BS 1
               .DA MY.IRQ.PROCESSOR
```

will cause the address of MY.IRQ.PROCESSOR to be stored in the IRQ Path table. The index into the table pointing to the entry used will be converted to an integer from 1 to 4, and stored at IRQ.NUM in the IOB. The purpose of this number is to allow you to later de-allocate the interrupt if you wish. A call and an IOB like this will de-allocate an interrupt:

```
          JSR $BF00
          .DA $41,IRQ.IOB
       ---
       ---
       IRQ.IOB .DA #1
       IRQ.NUM .BS 1   (filled in by program)
```

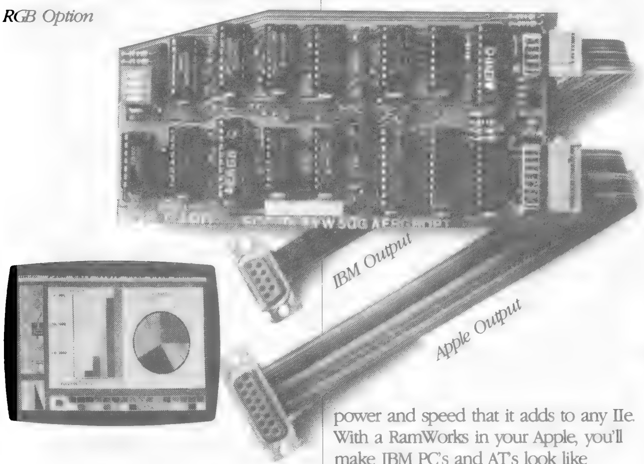
Note that the first byte if the IOB is different this time, because there is only one parameter rather than two. It is important to de-allocate, because otherwise a sneaky interrupt could occur which would cause ProDOS to branch after your program is gone.

Another way to de-allocate is to store zeroes directly into the IRQ Path table, but Apple warns against this practice. It is quicker and easier, though.

There may be more than one source of IRQ interrupts in a given system. For example, you may be using both a clock card and a modem, both with interrupts. ProDOS allows you to have separate interrupt handlers for each of them installed. When an IRQ occurs the first handler installed will be called first. If that handler determines the IRQ is its own, it should

Applied Engineering doesn't make you buy it, but if you want RGB output you're in for a nice surprise because the RamWorks RGB option offers better color graphics plus a more readable 80 column text (that blows away any composite color monitor). For only $129 it can be added to RamWorks giving you a razor sharp, vivid brilliance that most claim is the best they have ever seen. You'll also appreciate the multiple text colors (others only have green) that come standard. But the RamWorks RGB option is more than just the ultimate in color output because unlike others, it's fully compatible with all the Apple standards for RGB output control, making it more compatible with off-the-shelf software. With its FCC certified design, you can use almost any RGB monitor because only the new RamWorks RGB option provides both Apple standard and IBM standard RGB outputs (cables included). The RGB option plugs into the back of RamWorks with no slot 1 interference and remember you can order the RGB option with your RamWorks or add it on at a later date.

RGB Option



True 65C816 16 Bit Power.

RamWorks III has a built-in 65C816 CPU port for direct connection to our optional 65C816 card. The only one capable of linearly addressing more than 1 meg of memory for power applications like running the Lotus 1-2-3™ compatible program, VIP Professional. Our 65C816 card does not use another slot but replaces the 65C02 yet maintains full 8 bit compatibility.

Endorsed by the Experts.

A+ magazine said "Applied Engineering's RamWorks is a boon to those who must use large files with AppleWorks...I like the product so much that I am buying one for my own system." inCider magazine said "RamWorks is the most powerful auxiliary slot memory card available for your IIe, and I rate it four stars...For my money, Applied Engineering's RamWorks is king of the hill."

"I wanted a memory card for my Apple that was fast, easy to use, and very compatible; so I bought RamWorks."

Steve Wozniak, the creator of Apple Computer

Apple experts everywhere are impressed by RamWorks's expandability, versatility, ease of use, and the sheer power and speed that it adds to any IIe. With a RamWorks in your Apple, you'll make IBM PC's and AT's look like slowpokes.

Quality and Support of the Industry Leader

RamWorks III is from Applied Engineering, the largest, most well supported manufacturer of Apple peripherals and the inventor of large RAM cards for the Apple. With our 5 year no hassle warranty and outstanding technical support, you're assured of the most trouble free product you can buy.

It's Got It All

- 15 day money back guarantee
- 5 year hassle free warranty insures coverage no matter where you purchase
- Built-in super sharp 80 column display, (U.S. Patent #4601081)
- Expandable to 1 MEG on main card
- Expandable to 16 meg with expander cards, with NO slot 1 interference
- Can use 64K or 256K RAMs
- Powerful linear addressing 16 bit coprocessor port
- Automatic AppleWorks expansion up to 3017K desktop
- Accelerates AppleWorks
- Built-in AppleWorks printer buffer
- The only large RAM card that's 100% compatible with all IIe software
- RamDrive™ the ultimate disk emulation software included free
- Memory is easily partitioned allowing many programs in memory at once
- Compatible, RGB option featuring ultra high resolution color graphics and multiple text colors, with cables for both Apple and IBM type monitors
- Built-in self diagnostics software
- Lowest power consumption (U.S. Patent #4601081)
- Takes only one slot (auxiliary) even when fully expanded
- Socketed and user upgradeable
- Software industry standard
- Advanced Computer Aided Design
- Used by Apple Computer, Steve Wozniak and virtually all software companies
- Displays date and time on the AppleWorks screen with any PRO-DOS compatible clock
- Much, much more!

RamWorks III with 64K	$179
RamWorks III with 256K	$219
RamWorks III with 512K	$289
RamWorks III with 1 MEG	$369
RamWorks III with 1.5 MEG	$539
RamWorks III with 2 to 16 MEG	CALL
65C816 16 Bit Card	$159
RGB Option	$129
Optional Software:	
Pinpoint with RAM Enhancement Software	$79
VIP Professional	$219

RamWorks III. The industry standard for memory expansion of the Apple IIe.

ORDER YOUR RamWorks III TODAY. 9 a.m. to 11 p.m. 7 days, or send check or money order to Applied Engineering.

MasterCard, Visa and C.O.D. welcome. Texas residents add 5⅛% sales tax. Add $10.00 if outside U.S.A.

Applied Engineering™
P.O. Box 798, Carrollton, TX 75006
(214) 241-6060

process the IRQ and return with carry status clear. If not, the handler should return with carry status set. ProDOS will try giving the interrupt to each handler in turn, until one of them returns with carry clear. If none of them claim the IRQ, or if there are no processors allocated, "System Death" will occur: you will get error code $01, and a message to insert system disk and restart. It seems a nicer approach to unclaimed interrupts would be to count it, and continue processing. When the count exceeds some magic number, say 255, that would be the time to go through the agony of "System Death". I would also like to know the cause of "Death", if possible.

Lines 1400-1640 in the listing below show the code for allocating an interrupt. The code searches the IRQ Path table for an available entry (equal to 0000), and inserts the user's processor address in the first one found. If none are found you get error $25, INTERRUPT TABLE FULL. Actually only the high byte of each entry is checked, which means you cannot put an interrupt processing routine anywhere in page zero. The MLI call will allow you to do so, and it will even work, but if you later try to allocate another interrupt it will use the same table entry and clobber the first one. I suppose this is a bug in ProDOS, but not too likely to cause any problem because you are not likely to stick your code down in that page. Still, it COULD happen.

Lines 1660-1770 de-allocate an interrupt routine. If the interrupt index number is not in the range form 1 to 4, you will get error $53, BAD PARAMETER. Otherwise, the indicated entry will be zeroed.

When an IRQ interrupt occurs, if the status is such that interrupts are enabled, the processor status and the current PC-address will be pushed onto the stack; then processing will branch to the address currently at $FFFE and FFFF. What address is there will depend on which kind of Apple you are in, and whether ROMs or RAM are currently switched into the $D000-FFFF area. The original Apple II monitor and also the Apple II+ monitor vector IRQs into the $F8 monitor ROM area. A short routine there saves some registers and separates BRKs from IRQs (because they both share the same vector at $FFFE). IRQs then branch through another vector at $3FE and 3FF. The various Apple //e monitors vector IRQs and BRKs to an address at $C3FA, while the //c monitors send them to $C803.

When you boot ProDOS the installer/relocator code checks which kind of monitor you have. If your IRQ vector points anywhere below $D000, it assumes you have a "new style" monitor; if it points to anywhere between $D000 and $FFFF it assumes you have an "old style" monitor. The Apple II and II+ are old style, all others are new style. The installer/relocator stores a flag at $DFD8 so that the IRQ handler can tell what kind of machine it is in when an IRQ occurs later. This flag is shown at lines 2650-2710. In new style machines the vector found in ROM at $FFFE is copied into both Main and Auxiliary RAM banks at the same address, in case an interrupt occurs when RAM is switched on. In old style machines the address $FF9B is left

in the RAM vector, pointing to a special IRQ handler shown in lines 3060-3400 below.

The vector at $3FE,3FF is set up to point to IRQ.ENTRY, a short routine inside the MLI Global Page. This is shown in lines 3000-3040. Since no matter what kind of monitor is resident the IRQ eventually vectors here, I will start the explanation here. Lines 3020-3030 turn on RAM at $D000-FFFF, so that ProDOS is accessible. It also write enables the RAM, because the IRQ processing will be storing a value at $DFCE, which identifies the current owner of the $C800 space (lines 1970-1980).

Lines 1800-1870 save the registers in the MLI Global Page. If you are in an old style machine lines 1880-1950 will save the processor status and return address in the Global Page as well.

ProDOS wants to make it easier to write IRQ processors, so it also makes sure you can use the stack and some page zero. If there are not at least 128 bytes left on the stack it will pop off 16 bytes and save them in a special buffer; if there are at least 128 bytes left this step is skipped. Then lines 2070-2120 save zero page locations $FA through $FF in a special buffer. Your IRQ processor can use these six bytes without worrying about saving and restoring them. (If you need more, you will have to save-restore them yourself.) This is all nice, but it does add to the general overhead for processing interrupts, which is already burdensome.

Lines 2130-2320 sequence through the installed interrupt processors until one of them claims the IRQ. Lines 2330-2350 signal DEATH if none of the processors claim the IRQ.

If the IRQ is properly claimed, lines 2360-2410 restore the six zero page bytes; lines 2420-2500 restore the 16 bytes of stack space if they were previously saved. Lines 2520-2630 restore some registers and the $C800 space if you are in an old style machine, and in any case branch to the IRQ.EXIT routine in the MLI Global Page.

IRQ.EXIT, shown in lines 2800-2960, restores the correct kind of memory (RAM or ROM) and then executes an RTI instruction. In an old style machine if the IRQ happened during a time when the RAM was switched on, this will send control to IRQ.EXIT.OLD, shown in lines 3690-3740. In a new style machine, or in any machine if the ROMs were on when the IRQ happened, the RTI will go back to the control of the monitor; exactly where that is depends on which monitor. Normally BANK.ID.BYTE contains the value $01. If an IRQ occurs in an old style machine when RAM is switched on, it will be changed to $00 or $FF depending on which $D000 bank is selected. Lines 2930-2940 change it back to $01 after one either $00 or $FF is processed.

One advantage to having both IRQ.ENTRY and IRQ.EXIT in the MLI Global Page is that you could substitute your own code if you wish. If you want to reduce overhead, and you know that you will always be running in a specific monitor configuration, you

can patch in here. You could also patch in through the vector at $3FE, and avoid even more overhead. However, you would no longer be "standard".

I published a listing of lines 3070-3640 way back in December 1983, but I decided to include it here for completeness. This code is only used in an old style machine, and only when the IRQ occurs while RAM is switched on. The vector at $FFFE starts up the code at line 3140. The nonsense in lines 3140-3180 regarding location $45 makes sure we do not clobber the saved A-register. The old style monitor ROMs save the A-register at $45 rather than on the stack. This conflicts with use of the same location within both DOS 3.3 and ProDOS. QUESTION: Wouldn't it have been both easier and better to avoid using locaton $45 inside ProDOS? Kludge on top of kludge, if you ask me.

Lines 3290-3340 set up fake data on the stack for later use by an RTI instruction. Lines 3350-3380 do the same for an RTS instruction. Note that RTS requires an address with is one less than the actual address, while RTI requires the address un-modified. RTS pops the address, adds one, and branches; RTI pops the address and status, and branches without adding one. Line 3400 switches back to ROM. This means the next instruction will be executed from $FFCB in ROM, which is ALWAYS an RTS. Anyway it had BETTER be! If you ever make your own monitor ROM, be sure to leave an RTS here. (You will also need an RTS at $FF58, because a lot of I/O firmware expects that one is there.)

Lines 3420-3470 are executed in the old style machines if RAM is switched on when you hit RESET. That is, if you have the particular type of RAM card which leaves the F8 area switched on when you hit RESET. Many of them switch back to ROM when RESET occurs. Just in case, the code is here.

That about wraps it up. But it still leaves a lot of mystery in that part of IRQ processing which occurs inside the monitor ROMs. Each monitor version has its own unique code. The Apple II was simple, the II+ about the same. The three versions of //e and three versions of //c monitors of which I am aware are all mutually different. The IIgs is even more so. Perhaps in a future article we can rationalize them all.

```
                   1000 *SAVE MLI.IRQ
                   1010 *--------------------------------
40-                1020 PARM.PNTR        .EQ $40,41
42-                1030 COMMAND          .EQ $42
45-                1040 SAVE.A           .EQ $45
                   1050 *--------------------------------
07F8-              1060 CURRENT.ROM.SLOT .EQ $07F8   $C0 + Slot which owns $C800.
                   1070 *--------------------------------
BF09-              1080 CALL.SYSERR      .EQ $BF09
BF0C-              1090 CALL.DEATH       .EQ $BF0C
                   1100 *--------------------------------
BF56-              1110 SAVE.LOC45       .EQ $BF56   Used if in Apple II or II+
BF57-              1120 SAVE.D000        .EQ $BF57       ditto
                   1130 *--------------------------------
BF80-              1140 IRQ.PATH.1       .EQ $BF80   These are 0000 if not allocate
BF82-              1150 IRQ.PATH.2       .EQ $BF82   address of user IRQ handler
BF84-              1160 IRQ.PATH.3       .EQ $BF84   if allocated.
BF86-              1170 IRQ.PATH.4       .EQ $BF86
                   1180 *--------------------------------
BF88-              1190 IRQ.A            .EQ $BF88
BF89-              1200 IRQ.X            .EQ $BF89
BF8A-              1210 IRQ.Y            .EQ $BF8A
BF8B-              1220 IRQ.S            .EQ $BF8B
BF8C-              1230 IRQ.P            .EQ $BF8C
BF8D-              1240 BANK.ID.BYTE     .EQ $BF8D
BF8E-              1250 IRQ.RETURN       .EQ $BF8E,BF8F
                   1260 *--------------------------------
CFFF-              1270 IO.RESET.ROMS    .EQ $CFFF   De-select $C800 space.
                   1280 *--------------------------------
FEDF-              1290 IRQ.SV .EQ $FEDF thru $FEEE  (16 bytes saved from STACK)
                   1300 *--------------------------------
                   1310          .PH $DEF3
                   1320 *--------------------------------
                   1330 *   Handle $40 and $41 MLI calls
                   1340 *--------------------------------
                   1350 * (DEF3) DE57
                   1360 INTERRUPT.HANDLER
DEF3- 85 42        1370          STA COMMAND  Save in case anyone cares later.
DEF5- 4A           1380          LSR          $40 or $41, lsb into CARRY
DEF6- B0 2C        1390          BCS .5       ...$41 is DEALLOCATE
                   1400 *---$40 is ALLOCATE--------------
DEF8- A2 03        1410          LDX #3       FOR X = 3 TO 9 STEP 2
DEFA- BD 7E BF     1420 .1       LDA IRQ.PATH.1-2,X
DEFD- D0 16        1430          BNE .2       ...ALREADY ALLOCATED
DEFF- A0 03        1440          LDY #3       FOUND HOLE, INSTALL IRQ
DF01- B1 40        1450          LDA (PARM.PNTR),Y
DF03- F0 1A        1460          BEQ .3       BAD PARAMETER
DF05- 9D 7E BF     1470          STA IRQ.PATH.1-2,X
DF08- 88           1480          DEY
DF09- B1 40        1490          LDA (PARM.PNTR),Y
DF0B- 9D 7D BF     1500          STA IRQ.PATH.1-3,X
DF0E- 8A           1510          TXA          GIVE IRQ# TO CALLER
DF0F- 4A           1520          LSR          MAKE 3,5,7,9 INTO 1,2,3,4
DF10- 88           1530          DEY
DF11- 91 40        1540          STA (PARM.PNTR),Y
DF13- 18           1550          CLC          Signal NO ERROR
DF14- 60           1560          RTS
DF15- E8           1570 .2       INX          Next X
DF16- E8           1580          INX
DF17- E0 0B        1590          CPX #11
DF19- D0 DF        1600          BNE .1
DF1B- A9 25        1610          LDA #$25     "INTERRUPT TABLE FULL"
DF1D- D0 02        1620          BNE .4       ...ALWAYS
DF1F- A9 53        1630 .3       LDA #$53     "BAD PARAMETER"
DF21- 20 09 BF     1640 .4       JSR CALL.SYSERR   (NEVER RETURNS)
                   1650 *---$41 is DEALLOCATE------------
DF24- A0 01        1660 .5       LDY #1
DF26- B1 40        1670          LDA (PARM.PNTR),Y
DF28- F0 F5        1680          BEQ .3       ...0 is illegal value
DF2A- C9 05        1690          CMP #5       Must be 1,2,3,4
DF2C- B0 F1        1700          BCS .3       ...too large
DF2E- 0A           1710          ASL          DOUBLE FOR INDEX
DF2F- AA           1720          TAX
DF30- A9 00        1730          LDA #0       CLEAR THE ENTRY
DF32- 9D 7E BF     1740          STA IRQ.PATH.1-2,X
```

```
DF35- 9D 7F BF 1750        STA IRQ.PATH.1-1,X
DF38- 18       1760        CLC           Signal NO ERROR
DF39- 60       1770        RTS
               1780 *--------------------------------
               1790 *   If an IRQ occurs, we eventually get HERE.
               1800 *--------------------------------
               1810 IRQ.HANDLER
DF3A- A5 45    1820        LDA SAVE.A
DF3C- 8D 88 BF 1830        STA IRQ.A
DF3F- 8E 89 BF 1840        STX IRQ.X
DF42- 8C 8A BF 1850        STY IRQ.Y
DF45- BA       1860        TSX
DF46- 8E 8B BF 1870        STX IRQ.S
DF49- AD D8 DF 1880        LDA ENHANCE.FLAG
DF4C- D0 0C    1890        BNE .1        ...In a "new style" monitor
DF4E- 68       1900        PLA           ...In an Apple II or II+ monitor
DF4F- 8D 8C BF 1910        STA IRQ.P     Save P-reg and RETURN address
DF52- 68       1920        PLA
DF53- 8D 8E BF 1930        STA IRQ.RETURN
DF56- 68       1940        PLA
DF57- 8D 8F BF 1950        STA IRQ.RETURN+1
DF5A- 9A       1960 .1     TXS           Keep P-reg and RETURN on stack
DF5B- AD F8 07 1970        LDA CURRENT.ROM.SLOT   Save $C800 Slot
DF5E- 8D CE DF 1980        STA ROM.PAGE.BYTE
               1990 *---Save some stack, maybe-------
DF61- BA       2000        TSX           If in bottom half of stack,
DF62- 30 09    2010        BMI .3            then save 16 bytes of it.
DF64- A0 0F    2020        LDY #15       SAVE 16 BYTES FROM STACK
DF66- 68       2030 .2     PLA
DF67- 99 DF FE 2040        STA IRQ.SV,Y
DF6A- 88       2050        DEY
DF6B- 10 F9    2060        BPL .2
               2070 *---Save some page zero----------
DF6D- A2 FA    2080 .3     LDX #$FA      SAVE 6 BYTES FROM PAGE ZERO
DF6F- B5 00    2090 .4     LDA 0,X       $FA...FF
DF71- 9D F5 FD 2100        STA IRQ.SV-$FA+16,X
DF74- E8       2110        INX
DF75- D0 F8    2120        BNE .4
               2130 *---Call to first IRQ vector-----
DF77- AD 81 BF 2140        LDA IRQ.PATH.1+1
DF7A- F0 05    2150        BEQ .5        IRQ#1 EMPTY
DF7C- 20 D9 DF 2160        JSR IRQ.1     Try this IRQ level
DF7F- 90 23    2170        BCC .9        ...IRQ Claimed, Now Exit
               2180 *
DF81- AD 83 BF 2190 .5     LDA IRQ.PATH.2+1
DF84- F0 05    2200        BEQ .6        IRQ#2 EMPTY
DF86- 20 DC DF 2210        JSR IRQ.2     Try this IRQ level
DF89- 90 19    2220        BCC .9        ...IRQ Claimed, Now Exit
               2230 *
DF8B- AD 85 BF 2240 .6     LDA IRQ.PATH.3+1
DF8E- F0 05    2250        BEQ .7        IRQ#3 EMPTY
DF90- 20 DF DF 2260        JSR IRQ.3     Try this IRQ level
DF93- 90 0F    2270        BCC .9        ...IRQ Claimed, Now Exit
               2280 *
DF95- AD 87 BF 2290 .7     LDA IRQ.PATH.4+1
DF98- F0 05    2300        BEQ .8        IRQ#4 EMPTY
DF9A- 20 E2 DF 2310        JSR IRQ.4     Try this IRQ level
DF9D- 90 05    2320        BCC .9        ...IRQ Claimed, Now Exit
               2330 *---No IRQ vectors alive!--------
DF9F- A9 01    2340 .8     LDA #$01      Un-claimed Interrupt Error
DFA1- 20 0C BF 2350        JSR CALL.DEATH    (NEVER RETURNS)
               2360 *---IRQ PROCESSING COMPLETE------
DFA4- A2 FA    2370 .9     LDX #$FA      RESTORE $FA...FF
DFA6- BD F5 FD 2380 .10    LDA IRQ.SV-$FA+16,X
DFA9- 95 00    2390        STA 0,X
DFAB- E8       2400        INX
DFAC- D0 F8    2410        BNE .10
               2420 *---If saved, restore stack------
DFAE- AE 8B BF 2430        LDX IRQ.S
DFB1- 30 0B    2440        BMI .12       16 BYTES FROM STACK NOT SAVED
DFB3- A0 00    2450        LDY #0        RESTORE 16 BYTES TO STACK
DFB5- B9 DF FE 2460 .11    LDA IRQ.SV,Y
DFB8- 48       2470        PHA
DFB9- C8       2480        INY
DFBA- C0 10    2490        CPY #16
```

```
DFBC- D0 F7               BNE .11
                     *---Choose EXIT routine----------
DFBE- AD D8 DF       .12     LDA ENHANCE.FLAG
DFC1- D0 12               BNE IRQXIT     ..."New style" monitor ROMs
DFC3- AC 8A BF            LDY IRQ.Y      ...Apple II or II+ monitor.
DFC6- AE 89 BF            LDX IRQ.X
DFC9- AD FF CF            LDA IO.RESET.ROMS    Turn off $C800 bank
DFCC- AD 00 CF            LDA $CF00            Select Interrupted $C800 bank
                     * (DFCE) DF5E DFCF        (Hi-byte filled in)
DFCE-                ROM.PAGE.BYTE .EQ *-1     (Self-modifying code!)
DFCF- AD CE DF            LDA ROM.PAGE.BYTE
DFD2- 8D F8 07            STA CURRENT.ROM.SLOT
                     * (DFD5) DFC1
DFD5- 4C D0 BF       IRQXIT JMP IRQ.EXIT
                     *--------------------------------
                     * (DFD8) DF49 DFBE
DFD8- 00             ENHANCE.FLAG .HS 00 (Set to 01 by relocator if new
                     *                     type ROM is found)
                     *   ((If IRQ vector in ROM at $FFFE,F points below
                     *       $D000, it is "new type".  The "old type",
                     *       which is the original $F8 ROM in the Apple II
                     *       or that of the Apple II+, points to $Fxxx.))
                     *--------------------------------
                     * (DFD9-E2) DF7C DF86 DF90 DF9A
DFD9- 6C 80 BF       IRQ.1 JMP (IRQ.PATH.1)
DFDC- 6C 82 BF       IRQ.2 JMP (IRQ.PATH.2)
DFDF- 6C 84 BF       IRQ.3 JMP (IRQ.PATH.3)
DFE2- 6C 86 BF       IRQ.4 JMP (IRQ.PATH.4)
                     *--------------------------------
                          .PH $BFD0
                     *--------------------------------
                     *   IRQ ENTRY/EXIT CODE IN GLOBAL PAGE
                     *--------------------------------
                     IRQ.EXIT
BFD0- AD 8D BF            LDA BANK.ID.BYTE
                     IRQ.EXIT.1
BFD3- F0 0D               BEQ .2
BFD5- 30 08               BMI .1
BFD7- 4A                  LSR
BFD8- 90 0D               BCC .3
BFDA- AD 81 C0            LDA $C081    Switch on ROMs at D000-FFFF
BFDD- B0 08               BCS .3       ...ALWAYS
BFDF- AD 83 C0       .1   LDA $C083    Switch on RAMs at D000-FFFF
BFE2- A9 01          .2   LDA #1
BFE4- 8D 8D BF            STA BANK.ID.BYTE
BFE7- AD 88 BF       .3   LDA IRQ.A
BFEA- 40                  RTI
                     *--------------------------------
                     *   An IRQ interrupt comes here when it occurs
                     *       because of the vector at $3FE,3FF.
                     *--------------------------------
                     IRQ.ENTRY
BFEB- 2C 8B C0            BIT $C08B    Switch on and write-enable
BFEE- 2C 8B C0            BIT $C08B        RAM at D000-FFFF
BFF1- 4C 3A DF            JMP IRQ.HANDLER
                     *--------------------------------
                          .PH $FF9B
                     *--------------------------------
                     *   IRQ CODE FOR APPLE II AND II+ MONTOR ROMS
                     *       This code is used when IRQ happens while
                     *       the RAM at D000-FFFF is switched on (inside
                     *       an MLI call, for example) if we have the
                     *       "new style" monitor ROMs.
                     *--------------------------------
FF9B- 48             IRQ  PHA          SAVE A-REG
FF9C- A5 45               LDA SAVE.A       ALSO SAVE SAVE.A
FF9E- 8D 56 BF            STA SAVE.LOC45
FFA1- 68                  PLA          NOW PUT A-REG INTO SAVE.A
FFA2- 85 45               STA SAVE.A
FFA4- 68                  PLA          PEEK AT STATUS
FFA5- 48                  PHA
FFA6- 29 10               AND #$10     WAS IT "BRK"?
FFA8- D0 18               BNE .2       ...YES, LET MONITOR HANDLE IT
FFAA- AD 00 D0            LDA $D000    CHECK WHETHER D000 BANK 1 OR 2
FFAD- 49 D8               EOR #$D8     "CLD" OPCODE
```

```
FFAF- F0 02              BEQ .1        ...IN D000 BANK 1
FFB1- A9 FF              LDA #$FF      ...IN D000 BANK 2
FFB3- 8D 8D BF     .1    STA BANK.ID.BYTE
FFB6- 8D 57 BF           STA SAVE.D000
FFB9- A9 BF              LDA /IRQ.EXIT.OLD   PUSH FAKE "RTI" VECTOR
FFBB- 48                 PHA
FFBC- A9 50              LDA #IRQ.EXIT.OLD
FFBE- 48                 PHA
FFBF- A9 04              LDA #$04
FFC1- 48                 PHA
FFC2- A9 FA        .2    LDA /$FA41    PUSH FAKE "RTS" VECTOR INTO
FFC4- 48                 PHA                MONITOR ROM
FFC5- A9 41              LDA #$FA41
FFC7- 48                 PHA
                   CALL.MONITOR
FFC8- 8D 82 C0           STA $C082     SWITCH TO MOTHERBOARD ROMS,
                   *                   WHERE THERE IS AN "RTS" OPCODE
                   *-------------------------------
FFCB- AD D7 FF     RESET LDA RESET.VECTOR+1
FFCE- 48                 PHA           PUSH FAKE "RTS" INTO MONITOR
FFCF- AD D6 FF           LDA RESET.VECTOR
FFD2- 48                 PHA
FFD3- 4C C8 FF           JMP CALL.MONITOR
                   *-------------------------------
FFD6- 61 FA        RESET.VECTOR .DA $FA61   MON.RESET-1
                   *-------------------------------
                   IRQ.SPLICE
FFD8- 8D 88 BF           STA IRQ.A
FFDB- AD 56 BF           LDA SAVE.LOC45
FFDE- 85 45              STA SAVE.A
FFE0- AD 8B C0           LDA $C08B    FINISH WRITE-ENABLING RAM
FFE3- AD 57 BF           LDA SAVE.D000
FFE6- 4C D3 BF           JMP IRQ.EXIT.1
                   *-------------------------------
FFE9-                    .BS $FFFA-*  <<<EMPTY SPACE>>>
                   *-------------------------------
FFFA- FB 03        V.NMI      .DA $03FB
FFFC- CB FF        V.RESET    .DA RESET
FFFE- 9B FF        V.IRQ      .DA IRQ  (Replaced by relocator with
                   *                   the value from ROM vector if
                   *                   the machine has "new style" monitor.
                   *-------------------------------
                         .PH $BF50
                   *-------------------------------
                   *   LITTLE PIECE OF IRQ EXIT CODE USED WITH
                   *      OLD TYPE MONITOR ROMS
                   *-------------------------------
                   IRQ.EXIT.OLD
BF50- AD 8B C0           LDA $C08B    SWITCH RAM ON, D000 BANK 1
BF53- 4C D8 FF           JMP IRQ.SPLICE
                   *-------------------------------
```

$48 and "G" Strike Again!................Bob Sander-Cederlof

This is still a problem. When you use the monitor "G" command, it loads up the registers from $45 through $49 and then goes to the address you specified. Location $48 supplies the status, the P-register. Unfortunately, both DOS and ProDOS walk on location $48. Therefore it might have a meaningless value.

Worse than meaningless, it might be dangerous. It was for me a few days ago. I was debugging a program under ProDOS, and found that if I started it up with "800G" immediately after a SAVE command, it went completely crazy. The screen output was totally trashed. Scrolling turned into shuffling, the cursor jumped around the screen when typing in characters, and so on.

Why? Because $48 had a garbage value with bit 3 turned on, which turned on the decimal mode. Apparently SAVE stores something into $48, and the actual value depends on mysterious factors. Anyway, I woke up the next morning with the explanation of what was going on. I even remembered that I wrote an article about the problem in the June 1984 issue of AAL (pages 13 and 14).

One solution is to put a "CLD" instruction at the beginning of my program, just in case. Other, less secure fixes are to type "48:4 N 800G" instead of just "800G", or avoid using the G-command altogether by putting the code on a BRUNnable file.

A Bug in FID with 400K Volumes...........Bob Sander-Cederlof

Tom Weishaar sent me copies of some correspondence regarding the RamFactor boards and and Apple's FID program. If the name FID means nothing to you, it is the DOS 3.3 equivalent to the ProDOS FILER program, and comes on the Apple DOS 3.3 System Master Disk. It seems that if you use FID to transfer files to a 400K DOS 3.3 partition on the card that FID quits with the DISK FULL error long before the volume is really full.

I tried to reproduce the problem on my system. When I used FID, it failed just as stated above. The exact point at which FID gave up seemed to vary with the size of the files being copied. When I used plain DOS, using BSAVE or SAVE, I had no problem really filling the disk. I also tried it with my word processor, with no problems. Apparently the problem is in FID.

When I got the DISK FULL message from FID, I went to the monitor and took a look at the VTOC sector of the RamFactor drive. The VTOC buffer is located at $B3BB thru $B4BA after leaving FID. Sure enough, there were plenty of free sectors left, but they were almost all in sectors 0 thru 15 (the third and fourth bytes of each track's bit map) of each track. I went back into FID and experimented with trying to copy various size files. FID had no trouble copying files of fewer than 17 sectors, but on larger files it said DISK FULL again. It seems FID has a bug in the subroutine which looks for free sectors to COPY into.

I investigated a little more, with the help of a partial disassembly of FID which Bill Morgan did back in 1982. Sure enough, the subroutine to find the next free sector ignores the third and fourth bytes of each track's bitmap in the VTOC. Since FID allows the DOS File Manager to do part of the allocation, some of those sectors do get used. However, when the GET.FREE.SECTOR subroutine is called it will not use sectors 0-15 of a 32 sector per track disk.

Lines 1000-1210 in the listing which follows are the offending code from FID. As you can see, only two bytes of the VTOC are moved (lines 1090-1120). Lines 1130-1150 check if those two bytes are both zero, and if so move on to the next track.

My first attempt to fix the bug is shown in lines 1220-1430. I re-arranged the code somewhat, to free up the X-register so that I could write a loop which would move four bytes from the VTOC to the bitmap work area. The loop is in lines 1330-1390. This did not leave room for code to check whether all four bytes were zero or not, so I just left out that feature; it was not necessary anyway, but it did shave a few microseconds off the worst cases.

In my second attempt I found room for testing whether or not there are any free sectors in a track and avoiding the extra time it takes to look at the bitmap one bit a time. Lines 1440-1680 show the result.

```
                  1000 *SAVE S.PATCH.FID
                  1010 *--------------------------------
                  1020        .OR $11E0
                  1030        .TA $21E0
                  1040 *--------------------------------
11E0- 8A          1050        TXA          Original Buggy Code as Apple
11E1- 0A          1060        ASL               wrote it.
11E2- 0A          1070        ASL
11E3- A8          1080        TAY
11E4- B9 8A 19    1090        LDA $198A,Y
11E7- 8D 31 19    1100        STA BITMAP.OF.CURRENT.TRACK
11EA- B9 8B 19    1110        LDA $198B,Y
11ED- 8D 32 19    1120        STA $1932
11F0- D0 05       1130        BNE $11F7
11F2- 0D 31 19    1140        ORA BITMAP.OF.CURRENT.TRACK
11F5- F0 E3       1150        BEQ $11DA
11F7- 8E 93 13    1160        STX $1393
11FA- 8E 30 19    1170        STX FREE.TRK
11FD- AE 87 19    1180        LDX $1987    # OF SECTORS PER TRACK
1200- 8E 2F 19    1190        STX FREE.SEC
1203- 8C 92 13    1200        STY $1392
1206- D0 9C       1210        BNE $11A4
                  1220 *--------------------------------
                  1230        .OR $11E0
                  1240        .TA $31E0
                  1250 *--------------------------------
11E0- 8E 93 13    1260 PATCH1 STX $1393
11E3- 8E 30 19    1270        STX FREE.TRK
11E6- 8A          1280        TXA
11E7- 0A          1290        ASL          MULTIPLY BY FOUR
11E8- 0A          1300        ASL
11E9- A8          1310        TAY
11EA- 8C 92 13    1320        STY $1392
11ED- A2 00       1330        LDX #0       COPY 4 BYTES TO WORK AREA
11EF- B9 8A 19    1340 .1     LDA $198A,Y
11F2- 9D 31 19    1350        STA BITMAP.OF.CURRENT.TRACK,X
11F5- C8          1360        INY
11F6- E8          1370        INX
11F7- E0 04       1380        CPX #4
11F9- 90 F4       1390        BCC .1
11FB- AE 87 19    1400        LDX $1987    # OF SECTORS PER TRACK
11FE- 8E 2F 19    1410        STX FREE.SEC
1201- D0 A1       1420        BNE $11A4
1203-             1430        .BS 5,$EA    FILLER IS "NOP"
                  1440 *--------------------------------
                  1450        .OR $11E0
                  1460        .TA $41E0
                  1470 *--------------------------------
11E0- 8E 93 13    1480 PATCH2 STX $1393
11E3- 8E 30 19    1490        STX FREE.TRK
11E6- 8A          1500        TXA
11E7- 0A          1510        ASL          MULTIPLY BY FOUR
11E8- 0A          1520        ASL          ALSO CLEARS CARRY
11E9- A8          1530        TAY
11EA- 8C 92 13    1540        STY $1392
11ED- A2 FC       1550        LDX #$FC     256-4
11EF- B9 8A 19    1560 .1     LDA $198A,Y
11F2- F0 01       1570        BEQ .2       THIS ONE HAS NO FREE SECTORS
11F4- 38          1580        SEC          SIGNAL THERE WERE FREE SECTORS
11F5- 9D 35 18    1590 .2     STA BITMAP.OF.CURRENT.TRACK-$FC,X
11F8- C8          1600        INY
11F9- E8          1610        INX
11FA- D0 F3       1620        BNE .1       ...UNTIL 4 BYTES MOVED
11FC- 90 D9       1630        BCC $11D7
11FE- AE 87 19    1640        LDX $1987    # OF SECTORS PER TRACK
1201- 8E 2F 19    1650        STX FREE.SEC
1204- D0 9E       1660        BNE $11A4
1206- EA          1670        NOP          FILLER IS "NOP"
1207- EA          1680        NOP
                  1690 *--------------------------------
192F-             1700 FREE.SEC   .EQ $192F
1930-             1710 FREE.TRK   .EQ $1930
1931-             1720 BITMAP.OF.CURRENT.TRACK .EQ $1931 ... 1934
                  1730 *--------------------------------
```

To install the patch, first assemble it as shown. This will leave a copy of the original code starting at $21E0, of PATCH1 at $31E0, and PATCH2 at $41E0. Then BLOAD your copy of FID, and check the starting address and length by looking at AA60.AA61 for the length and AA72.AA73 for the starting address. In my copy, the starting address was $803 and the length was $124F. Using the monitor, verify that the original code at $11E0 is the same. If you have a different version of FID, this is the time to find out!

```
:   (type in the source code as shown)
:ASM

:BLOAD FID
:$AA60.AA61 AA72.AA73
AA60:4F 12
AA72:03 08        (this is what Apple displays)
:$11E0<21E0.2207V
```

If no discrepancies are reported, you can copy either PATCH1 or PATCH2 into place and BSAVE the result.

```
:$11E0<31E0.3207M   (for PATCH1)
          or
:$11E0<41E0.4207M   (for PATCH2)

:BSAVE FID.PATCHED,A$803,L$124F
```

A patched version of FID is included on the monthly and quarterly disks for this issue.

Apple Assembly Line (ISSN 0889-4302) is published monthly by S-C SOFTWARE CORPORATION, P.O. Box 280300, Dallas, Texas 75228. Phone (214) 324-2050. Subscription rate is $18 per year in the USA, sent Bulk Mail; add $3 for First Class postage in USA, Canada, and Mexico; add $14 postage for other countries. Back issues are $1.80 each (other countries inquire for postage). A subscription to the newsletter and the Monthly Disk containing all source code is $64 per year in the US, Canada and Mexico, and $87 to other countries.